Compiler Principles, PS5

Top-Down Parsing - Transforming Grammar into LL(1)

Sometimes grammars that are not LL(1) but their corresponding language is LL(1). Meaning, there is a different grammar for this language which is LL(1).

In the general case we don't know to find this LL(1) grammar whenever it exists. What we do know, is apply some algorithms that will: either give us an equivalent LL(1) grammar OR the algorithm will fail and enter an endless loop. The good news is we will never receive a wrong output!

Epsilon Derivation Removal

Our first Assumption on grammars is that there are no epsilon derivations, for each epsilon derivation A → ε and for all Derivations containing A on the right side B → αAβ , we add the rule B → αβ. Now we can remove the derivation.

The Only Non-Terminal which can have epsilon derivations is the Starting Non-Terminal (S).

Example 1
S → AB
A → Aa | ε | Cc
B → C | b
C → Cv | ε | w

Is transformed to:
Removing c → ε
S → AB
A → Aa | ε | Cc | c
B → C | ε | b |
C → Cv | v | w

Removing B → ε
S → AB | A
A → Aa | ε | Cc | c - (stays the same)
B → C | b
C → Cv | v | w - (stays the same)

Removing A → ε
S → AB | B | A | ε
A → Aa | a | Cc | c
B → C | b
C → Cv | v | w
Example 2
S → AB
A → AAB | ε | a
B → CDC | A | b | ε
C → c | ε
D → d

Removing A → ε
S → AB | B
A → AAB | AB | B | a
B → CDC | A | ε (redundant) | b | ε
C → c | ε
D → d

Removing B → ε
S → AB | A | B | ε
A → AAB | AA | AB | A (redundant) | B | ε
(We do not include ε because it already appeared and was removed)
B → CDC | A | b
C → c | ε
D → d

Removing C → ε
S → same
A → same
B → CDC | DC | CD | D | A | b
C → c
D → d

Left Recursion Elimination

Direct Left Recursion

D → Dy
D → x

Indirect Left Recursion

A → BC
B → DE
D → FG
F → AH
A → a
B → b
C → c
D → d
Removing direct left-recursion:
In the general case we replace all derivations \( \alpha \) from \( A \) that are not immediate left recursive with the Derivations \( A \rightarrow \alpha A' \), and all derivations that do begin \( A \rightarrow A\beta \) with the Derivation \( A' \rightarrow \beta A' \), and add the derivation \( A' \rightarrow \varepsilon \)

Suppose we have the productions

\[
A \rightarrow \alpha a | b
\]

We can replace them by

\[
A \rightarrow bA' \\
A' \rightarrow aA' | \varepsilon
\]

where \( A' \) is a new non-terminal.

Removing indirect left-recursion is done in the following manner:

1. Enumerate all non-terminals with \( S \) first, then starting with \( S = A_1 \) and continuing to \( A_n \).
2. For each derivation \( A_i \rightarrow A_j \beta \):
   2.1) If \( j < i \), replace \( A_j \) with all possible derivations \( w, A_j \rightarrow w \). (Substitute \( A_j \) in \( A_i \))
   2.2) Eliminate direct left-recursion for \( A_i \).

Example:

\[
S \rightarrow C | a \\
C \rightarrow Dd | c \\
D \rightarrow Cc | d
\]

Let: \( A_1 = S, A_2 = C, A_3 = D \).

Remove indirect recursion in \( A_3 \) (D).

\[
S \rightarrow C | a \\
C \rightarrow Dd | c \\
D \rightarrow Ddc | cc | d
\]
The productions are now:

S → C | a
C → Dd | c
D → ccD' | dD'
D' → dcD' | ε

No left recursion!

**Factorization**

Even Without Left-recursion, A Grammar might still not be LL(1):

S → bA
S → bB
A → a
B → c

Another type of transformation that has to be performed to the grammar productions in order to make them LL(1) is factorization.

Consider:

P → aPb
P → aPc
P → d

Which is not LL(1).

Following transformation is made to remove the problem:

P → aPX
X → b
X → c
P → d

in which “aP” has been factored out and X is new non-terminal.

Another example of factorization:

P → abQ
P → acR
Transformed into:

\[ P \rightarrow aX \\
X \rightarrow bQ \\
X \rightarrow cR \]

Sometimes we need to replace a non-terminal with all its derivations:

<table>
<thead>
<tr>
<th>Derivations</th>
<th>DSS</th>
</tr>
</thead>
<tbody>
<tr>
<td>( S \rightarrow Ax )</td>
<td>{a, b}</td>
</tr>
<tr>
<td>( S \rightarrow Br )</td>
<td>{a, b}</td>
</tr>
<tr>
<td>( A \rightarrow aA )</td>
<td>{a}</td>
</tr>
<tr>
<td>( A \rightarrow b )</td>
<td>{b}</td>
</tr>
<tr>
<td>( B \rightarrow bB \mid a )</td>
<td>{b}, {(a)}</td>
</tr>
</tbody>
</table>

\[ S \rightarrow aAx \quad \{a\} \]
\[ S \rightarrow bx \quad \{b\} \]
\[ S \rightarrow bBr \quad \{b\} \]
\[ S \rightarrow ar \quad \{a\} \]
\[ A \text{ and } B \text{ are the same} \]

\[ S \rightarrow aY \quad \{a\} \]
\[ S \rightarrow bZ \quad \{b\} \]
\[ Y \rightarrow Ax \mid r \quad \{a, b\}, \{r\} \]
\[ Z \rightarrow x \mid Br \quad \{x\}, \{a, b\} \]
\[ A \text{ and } B \text{ are the same} \]

Example in which factorization isn't successful:

| \( P \rightarrow Qx \) | \{s, q\} |
| \( P \rightarrow Ry \) | \{s, r\} |
| \( Q \rightarrow sQm \) | \{s\} |
| \( Q \rightarrow q \) | \{q\} |
| \( R \rightarrow sRn \) | \{s\} |
| \( R \rightarrow r \) | \{r\} |

\[ P \rightarrow sQmx \quad \{s\} \]
\[ P \rightarrow qx \quad \{q\} \]
\[ P \rightarrow sRny \quad \{s\} \]
\[ P \rightarrow ry \quad \{r\} \]
\[ Q \rightarrow sQm \quad \{s\} \]
\[ Q \rightarrow q \quad \{q\} \]
\[ R \rightarrow sRn \quad \{s\} \]
\[ R \rightarrow r \quad \{r\} \]

Next, following factorization can be performed:

| \( P \rightarrow sP' \) | \{s\} |
| \( P \rightarrow qx \) | \{q\} |
| \( P \rightarrow ry \) | \{r\} |

As far as \( P \) is concerned we eliminated intersections in the DSS:

\[ P' \rightarrow Qmx \quad \{s, q\} \]
\[ P' \rightarrow Rny \quad \{s, r\} \]
\[ Q \rightarrow sQm \quad \{s\} \]
\[ Q \rightarrow q \quad \{q\} \]
This grammar still isn't LL(1). In fact, there is no equivalent grammar that is LL(1), and the language defined by the grammar is not LL(1). If we continue and make factorization on P', we'll get some P'' which needs to be factorized and etc. Here we see how the factorization algorithm enters infinite loop.

Finding equivalent LL(1) grammar

Given some grammar G that is not LL(1), the simple algorithm for finding equivalent LL(1) grammar is:

0. Ensure Epsilon Derivations are removed
1. Left recursion elimination
2. Factorization

As mentioned above sometimes this algorithm may enter infinite loop and not find new LL (1) grammar.